CSE 4705 Final Project

Reed Kroll

Noah Pacik-Nelson

Tyler Nguyen

Matt Rumbel

13 December, 2019

A Series of Unfortunate Events, Book 14: Creating KonaneZero

**Preface**

In order to save the reader time, documentation on running the AI on the Artemis server will be provided here. To use KonaneZero and a completely untrained neural network as a player, run the “artemis\_connector.py” file. To use the min-max search AI, run the “minmaxConnector.py” file. Each of these has a pre-set username and password which will automatically be used to sign in. If you would like to change the sign-in information, modify the variables uName and password in the startGame() function. If you would like to play either AI against itself, simply run either the “dup-artemis\_connector.py” or “dup-minmaxConnector.py” file. These are exact duplicates of their respective counterparts; the only difference being the username and password for sign in.

**Introduction**

To design a konane artificial intelligence player, our team first attempted to utilize open source code from the AlphaZero AI and repurpose it. AlphaZero is the current best AI for playing multiple different types of games. Code for AlphaZero exists in an open GitHub repository (see references 1) which contains code for playing multiple games such as connect four, tafl, othello, and others. To generate an AI that plays konane, our team repurposed the code from the tafl and othello games to follow the rules of konane. After fully repurposing the code, our team found that it was not feasible to train the neural network on any obtainable run instance, in the time available before the tournament. Our team had to then change our code to use a min-max tree search, rather than the neural network, to play konane. This document will discuss, in depth, how all relevant code works, what our team changed to fit our needs, our attempts at training a neural network, and how we later refit the Artemis connector to use a min-max search rather than the neural network.

***Repurposing AlphaZero***

Our team initially began by repurposing open source code for the well-known AI, AlphaZero. We were able to modify the game tafl’s functions in such a way that it followed the konane set of rules but returned all functions in the same manner. This removed the need to redesign the entire system and focus on coding the game logistics only. We dubbed this repurposed code: KonaneZero.

**Base Code**

In this section we will discuss the functionality of all code which was not changed for the purpose of this project. These files control the training of the AI’s neural network for whichever game it is told to play.

main.py

The main.py file specifies the parameters for training the neural net and calls a Coach object to do the actual training. To better describe what each of the parameters does, we will discuss the basis for training a neural network in reference to each of the variables contained in the args dictionary (code comments reference which variable is linked to each parameter). When training a neural network, an iteration is what can be considered a single attempt at improving the neural network. An iteration begins by playing several games against itself using its current neural network; these games are called episodes. Each move made in an episode is generated by a monte-carlo search which plays certain moves to the end of the game and returns the optimal move. Each move made during each of these games is stored in memory. The program will also prohibit itself from storing too many of these game states in a queue to save memory. When all games have been played and stored, the program will use these games to train its neural network. By propagating through each of the game states in each game, the neural network will assign evaluation values to game states based on if it won or lost the game it is currently looking at. Multiple of these epochs are performed for the neural network to pick optimal weighting. The epochs variable is not contained in the main.py file, and will be discussed later, but is relevant to the overall operation of the neural network. When all epochs have been computed, the optimal neural network will be chosen and then forced to play a game against the old version of its neural network. If the new version of the neural network wins a certain threshold amount of the games played against the old neural network, the new neural network will be accepted as the better and current one to use. If the new neural network does not win this threshold amount of games, it will be rejected, and the old neural network will stay in use. This entire process will take place for each iteration of the program.

The main.py file will use all of these parameters for training a neural network and pass them into a coach object which it then tells to go do all of the work for it. The main.py file is what needs to be run for this program to execute, but the Coach object is what does all of the operations.

Coach.py

Coach.py contains the core neural net training loop. The Coach class executes the self-play and learning portions. The Coach class uses the functions defined in Game and NeuralNet, and takes additional arguments from our main.py, such as the number of complete self-play games to simulate during a new iterations, number of games our Monte Carlo Tree Search to simulate, number of games to playing (against itself) to determine if a new neural net will be accepted, threshold win proportion values for to determine whether a new neural net will be accepted, and a few others. There are several important methods within our Coach class.

First, we have our executeEpisode method. This method executes one episode of self-play, starting with player 1. As the game is played, each turn is added as a training example to trainExamples, which can grow rapidly. The game is played until the game ends. After the game is complete, the outcome of the game is used to assign values to each example in trainExamples. This function takes in only class variables and returns a large list of training examples.

Next, we have our learn method. Our learn method performs a specified number of iterations with a specified number of episodes of self-play in each iteration. After every iteration, we retain the neural network with examples in trainExamples (which has a maximum length of maxlenofQueue to cap memory usage). We then pit the new neural network against the best previous neural net to determine which model has performed better.

The last few methods of our coach class have to deal with keeping track of our learning process. Our getCheckpointFile method is used for reading a checkpoint file (in case of system crash and other unforeseen circumstances). Next, our saveTrainExamples method is used to save our current training examples (in case of system crash and other unforeseen circumstances). Lastly, our loadTrainExample method is used to load our current training examples (in case of system crash and other unforeseen circumstances).

MCTS.py

The MCTS.py is a Python implementation of the Monte Carlo Tree Search. The Monte Carlo Tree Search is a heuristic search algorithm for various kinds of decision processes. Our team used the MCTS that had already been implemented in the code for which we are repurposing since we know it to be working for the other games already implemented.

The Monte Carlo tree search works similar to minimax; however, it uses less computing power. Instead of expanding every possible state to a certain depth, the search will expand one layer, and then choose a state to continue expanding on based on the evaluations it has assigned. This removes the computing power needed to expand the entire tree and focuses the power on areas we know to be the potential best options. With each of these best possible options, MCTS will play the game out to completion. The argument that tells the class how many of these good options to explore is found in main.py. After a lot of training, the neural network will learn how to properly value different moves on its own.

Game.py

Game.py acts as an abstract class that provides a set of methods that needs to be implemented in KonaneGame.py. This class is defined due to the versatility it brings to the creation of many other different board games. We were able to find this implementation in other games, and thus implemented the same way for Konane.

Arena.py

The Arena is used to pit the most recent checkpoint of the neural network against the new version generated from the most recent set of played games. Arena is called at the very end of an iteration in Coach.py and gives the Arena players in the form of lambda functions. Each function calls a neural network from either the last checkpoint or most recent neural network and returns the most valuable move at a given board state. When arena is playing a set of games, these functions will be called for every move and therefore will return the best possible move for the respective neural network. Arena will play a set of games, half with one player starting and half with the other player starting and keep track of how much each player won, then returning these statistics. If new neural network has won a specified percentage of the games, it will be accepted as the new best neural network and written to the best.pth.tar file.

KonaneNNet.py

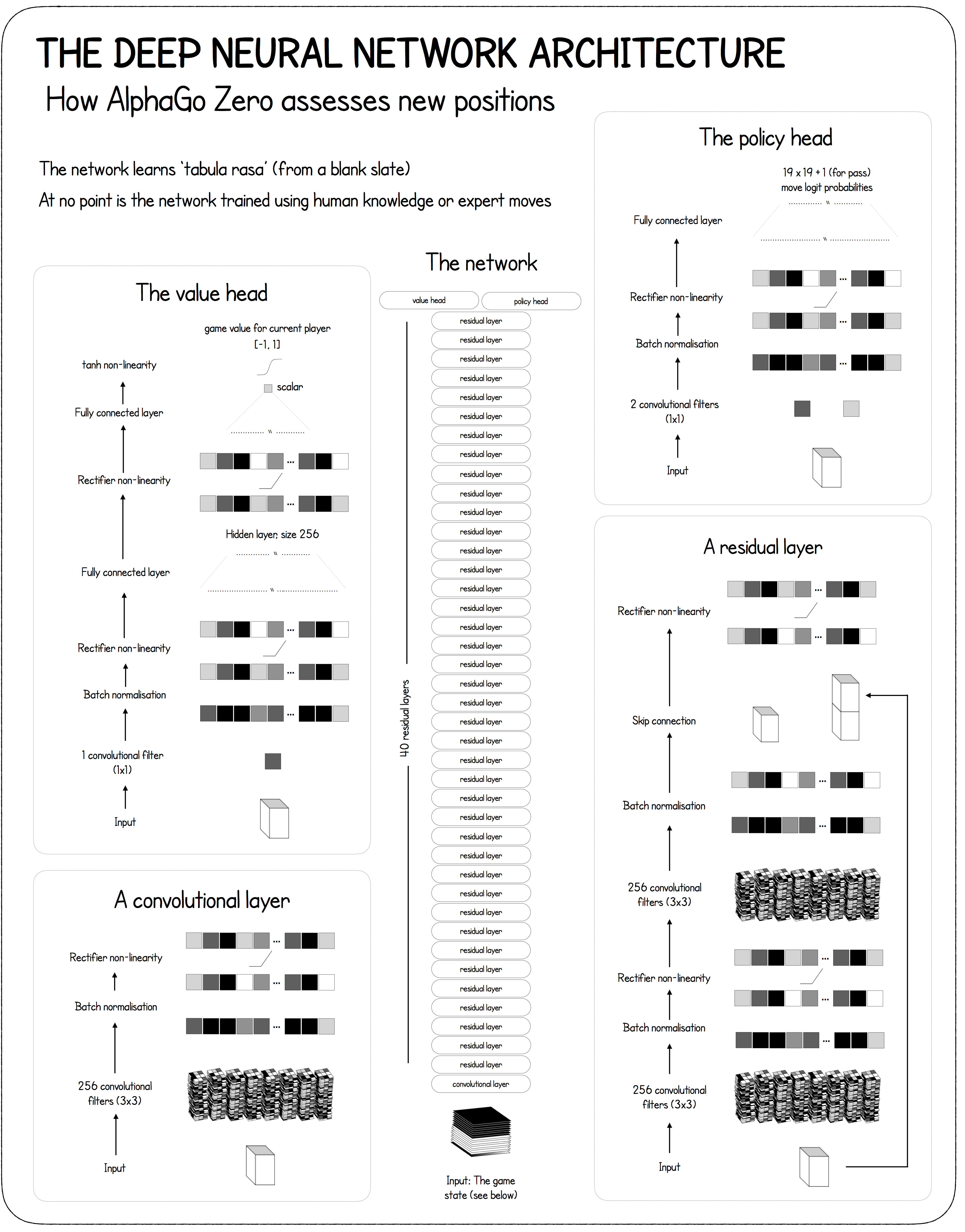
This file implements the modified architecture of AlphaZero used in the library. These two files (KonaneNNet.py and NNet.py) can be found in the folder of the particular machine learning framework you want to use. Currently, we’ve implemented solutions both in the pytorch and keras frameworks.

The architecture used by AlphaZero (see next page) to play go, chess, shogi, and other games follows a fairly consistent structure. A set of convolutional residual layers are stacked on top of each other, 40 layers deep, creating an incredibly complex neural network. This large stack then splits into two different heads, a single scalar quantity known as the value head dictates the learned value of the input board state, and the pi head, a vector the length of the total action size, which values every single possible and impossible move for both sides. Legal move values are then calculated by computing an element-wise multiplication of the pi vector and the vector of legal moves. The size of both of these (in the case of this Konane game) is 18^4, for the startX, startY, endX, and endY positions of a move.

The architecture we used was much simpler, keeping the idea behind the AlphaZero architecture, but greatly reducing the overall complexity. The architecture defined in the file KonaneNNet.py has 4 convolutional layers (we tried to implement 8 but saw diminishing return) with 2 feedforward layers at the end of the stack. Finally, the stack splits in two like AlphaZero, producing the value and pi heads.

NNet.py

This file is a framework specific manager of the KonaneZero neural network. A series of arguments at the top define things like epochs, dropout rates, and batch size. Inside the wrapper, functions save\_checkpoint and load\_checkpoint save and load neural net weights into files to ensure the durability of the program. The predict function runs the neural net on the input board state, returning the two heads previously discussed. The train function takes the example game states created through self-play in Coach.py and uses them to train the value of the state as well as the weights of each move.



**Konane Code**

The Konane game files are modified from the tafl and othello game files to fit the rules of playing konane. By modifying the othello and tafl game files, which we know work properly, we were able to rewrite the functions such that they returned values in the same form but perform the work differently.

For the game to learn from its played games, a neural net will be utilized with the pytorch library. The structure of the utilized neural net has also been repurposed from the code we are using.

KonaneGame.py

The KonaneGame object controls game-specific functionality to konane by using functions defined in the Board object in KonaneLogic.py. To avoid excess, the functionality of the KonaneGame object will be summarized rather than describing individual functions. The general purpose of the KonaneGame object is to find and return valid moves that are possible to make given a current game state. Whenever a move needs to be generated or executed, a new board object is created which then either finds a move using the current board state or modifies it by performing a given move. KonaneGame is also responsible for reporting when the end of the game has occurred which it does by, again, creating a board object and using functions provided by KonaneLogic. A vital part to KonaneGame is also the ability to return canonical forms which allows for the game to alternate players when it plays against itself. This was functionality that had been implemented via a different method in the original tafl game and had to be changed in our code by inverting the colors of pieces after every move. This allows us to use a single player object to play the game as it removes the need for two different objects to identify different colors of pieces as their own. One last thing to note is how the game represents itself visually for the neural net and internally for itself. You will learn in the next section that for ease of game computations, pieces are represented by 1, -1, and 0 for black, white, and no piece respectively. However, at the bottom of KonaneGame.py has a display function which translates the colored pieces to ‘X’ and ‘O’ with ‘-‘ representing an empty space. This is done to make it easier to interpret game states as the AI plays against itself; it is not used in the final version of the program but is very useful for our troubleshooting and ensuring the game was working properly.

KonaneLogic.py

The Board object in KonaneLogic.py is what contains all functionality that the KonaneGame object uses to process the game. We will discuss these functions more in depth than what was done in KonaneGame.py as they are what KonaneGame is based on.

The board initializes itself by fillings its rows with alternating ‘1’ and ‘-1.’ As was mentioned in the KonaneGame.py section, representing the board in this manner simplifies calculations for the program over, most importantly representing its canonical form as we discussed earlier.

The getLegalMoves function will return a list of all legal moves for the given player color. Since konane has special move cases for the first two turns, this function utilizes the \_isBoardFull function to check at what point the game is. \_isBoardFull will return either 0 when no pieces have been removed, 1 when one piece has been removed, or 2 when two or more pieces have been removed. If \_isBoardFull returns a 0 or 1, the game will make the special case moves for the first and second turn respectively and otherwise make a normal move using \_getValidMoves.

The \_getFirstMoves function returns a list of all the known legal moves allowed when starting the game. Since we are not actually moving a piece, but removing one, you will see that the start and end x and y coordinates are identical for each of the moves returned. This functionality will be talked about more in executeMove, however for now it is only necessary to know that it is easier to pass the duplicated coordinates for the purpose of using a single function for executing a move.

The \_getSecondMoves returns a list of all known legal moves when any of the starting legal moves has already been made. This function returns moves identically to \_getFirstMoves for the same reasons. This function identifies all possible second pieces by adding a move in one cardinal direction from the current position. After doing so, it then loops over each of these moves and ensures that none of their coordinates are out of bounds. It then returns all possible pieces to move after they have been verified as legal.

The \_getValidMoves function returns a list of all valid move. Each valid move is represented by a list containing the starting and ending x and y coordinates of a move. The function begins by setting up a standard iteration over the entire board and making sure that the current piece belongs to the player before continuing. After this there are four different for loops which calculate all possible moves in each cardinal direction. Each of these loops works by continually identifying if a jump in its corresponding direction is legal. A legal jump is when the piece next to the current location is of the opposing players color and the space two from the starting position is empty. If a jump is legal in a given direction, the move’s coordinates are stored into the legal\_moves array and jumps of longer length are checked for. If a jump is not legal, the loop is broken because a longer jump cannot be possible. Once all four of these for loops have been executed, we will have all possible legal moves at a given position.

The execute\_move function is the first step before a piece is actually moved. This function will first check if a board is full. If the board is full and the piece to remove is not the same as the given color, it will invert the board in order to orient the pieces correctly. The function then calls movePiece which carries out the actual modification of the board using the given move.

Lastly, \_movePiece takes a known legal move and changes the game state based off this. First, the function checks to see if the move is trying to remove a piece, i.e. the start and end coordinates are identical. If they are, we can remove just this piece and then return 1, the number of pieces “captured.” If a normal move is being made, the starting position is set to blank and the end position is set to the current players color. A list the positions of all captured pieces is then generated by \_getCaptures and \_movePiece then sets all of these positions to 0. When the function has finished removing the pieces, it will return the total number of pieces removed, which in this case is the length of the list returned by \_getCaptures.

**Learning Over Time**

When our game object finds a new neural net that performs better than its previous version, it will set this neural net to its new checkpoint for backup reasons and write the neural net to a file “best.pth.tar”. Storing the neural net in a load-able file will allow us to start up the neural net from the point at which we had trained the best possible version of itself. In order to load a previously generated neural net, the code will look to the args dictionary in main.py where there is a “load\_model” flag with a “load\_folder\_file” path. If the flag is set to “True”, the nnet will load in what is stored in the best.pth.tar file and then continue its learning process as normal. On a first run, there will be no file for the neural net to load as it has done no training yet. When a generated neural net is found to be better than its predecessor, the Coach object will write the neural net’s weights into the best.pth.tar file in a generated “temp” folder which can then be used to load at a later point. A sample temp folder has been included in the zip file, but has been relabeled as it is for a 12x12 Konane board.

**Problem Points During Development**

Over the course of our project we ran into five major issues which greatly slowed development: single piece playing, running computations on GPU, starting the game, finding valid moves late-game, and training time.

After repurposing the AlphaZero code to follow the rules of konane, we found that it was initially only moving one color piece. This may sound like a simple bug to fix, however solving such an issue was not as easy when using code written by another person. Our group had to sit down and go over the code in great depth in order to fully understand how variables were being passed between objects and functions. After over a day of analyzing the code we were able to identify that the source of error was a simple multiplication statement in which we had failed to change the orientation of the board for different players. This was a very small issue to fix however it ended up taking a lot of our time as we had not yet gotten to deeply understand the code at this point in our development. While this was annoying to work through, it ended up forcing us to learn a lot more about the code which inevitable helped us with our work later on.

The next major issue our team faced was getting the program to run on a GPU. While running some initial training, we found the program was using 100% of the CPU power on our computers and was still running very slow. On the group member’s computer with the strongest CPU it still took two and a half hours to do a single iteration of ten self-played games with 10 epochs. This was not a realistic amount of time to spend on an iteration, so we needed to look-into running the file on our GPU. Luckily the code came pre-built with a “cuda” flag that allows us to use our GPU on the file. The issue came when installing dependencies for this to work. Like the first issue, the solution was simple: download necessary python packages. While this is easy enough, finding out which packages we needed proved to be a real issue. We downloaded so many different packages that it would probably still be hard to tell another person exactly what they needed to install for the file to run properly. Once the proper packages had been installed, we then found that only about 0.2% of our GPU was being used for the file execution. This was extremely surprising as we were expecting to see something close to 100% like the CPU was using. This caused us to suspect the program was not actually using the GPU, however through more investigating we found this was just how much power the file used. While this may seem like a small amount of power, it increased run time dramatically and overall helped to speed up training time.

The third issue we encountered was generating incorrect starting game states. When testing gameplay, we found that the AI was making invalid starting moves in a significantly large amount of games. This was later isolated to knowing that the second move specifically was what was not being made correctly. At first, we could not find the issue in our code as it seemed that everything had been programmed correctly. However, after further inspection we found that we were removing items from a list and continuing to iterate steps in its indices, meaning we could sometimes “jump over” an invalid move and not remove it. When this bug was being fixed, our group was somewhat strapped for time so the current solution checks for all possible valid start spaces and then generates all the locations next to it. This is not an eloquent solution but suits our needs for the purposes of this project.

One of the final issues we encountered was the AI not finding moves at the end of the game when their opponent had run out of moves, but they had not. When testing the AI on the Artemis server against itself, we repeatedly ran into an issue where our opponent would be out of moves and we would be asked to make one, but the AI returned an error despite having moves available. Similar to the issue with making a valid second move, this issue arose very late into our development. This was one of the issues that took the longest to resolve because it required a deeper understanding of the code we had previously never modified. After a long time of tracing functions and variable usage, we identified that when the Monte Carlo tree search was looking through the current board state, it was not identifying any of the available moves and therefore setting their move values to 0. This resulted in a list of no possible moves to make which obviously raised an error. To further test the issue, we recreated a board which we knew was in one of these “error states” and ran separate methods to check the valid moves left. Through doing this we found that our basic game logic could identify the remaining valid moves, but the tree search could not. Our work-around to this issue was essentially a catch-case for this. Our modified code checks to see if an array of all 0’s is returned by the Monte Carlo search and executes a new set of code if it does. When this new block of code executes, it will search the board using our valid move search rather than the Monte Carlo search and put all these moves into the same hash table it uses normally. The only difference in this code is that all the found moves will be values the same, essentially meaning one will be chosen at random. This is more of a negative than a positive because it could cause our AI to not choose a move which gives the opponent another move, thus progressing the game rather than ending it. While this is not optimal, it was the best solution we could generate given the small timeframe we had to fix it.

The final issue we encountered with this project was that training the neural network took an immense amount of time and memory and this was the ultimate cause for our need to scrap the neural network altogether. There are several root issues with why training was so intense: the game space is too large, and we do not have enough time. The game space for an 18x18 board is ridiculously large. Games take over 200 moves to finish which not only takes a lot of time to complete but takes a lot of memory to store. Given that we are storing every move of each played game for the neural network to train off, our training examples would quickly build up during an iteration. We realized that by reducing the argument maxlenOfQueue in main.py, we could greatly reduce the amount of storage used during initial self-play. The harder problem arose when trying to make the game reduce its memory usage when playing against the old version of its neural network.

The memory usage during self-play in Arena was an issue our team could ultimately not resolve. Our team found that during arena play, every single game played would be stored in memory never released. This was an issue because (1) memory usage built up extremely fast and (2) we did not want these games to be stored in the first place. It made sense for our trainer to store games during initial self-play because this is how we generate training examples. However, during arena play we found that, despite no variables being stored, every game was being permanently stored in memory. Our team was able to isolate this issue specifically to line 46 in Arena.py. In this line, a game calls the lambda function that initialized the Arena object to return the best possible move with the current game state. For still unknown reasons this line of code was causing a memory to be stored with every move in the game. Our team attempted to fix this issue by directly deleting the “action” variable created in this line and then collecting garbage memory, but this still did not fix the issue. This left us in a position where even when running our code on a 52 GB Google Cloud instance, we would still not be able to get through even one iteration of training on standard parameters. We were left with two choices: try to train the neural network on extremely small training parameters or repurpose the entire game to use min-max search rather than a neural network. We ultimately decided on the latter as training the neural network on small iteration parameters would only generate a model which over-fit our examples and would most likely play poorly against other AI.

**A KonaneZero Retrospective**

While we were never able to train the KonaneZero neural network on a full 18x18 board, our group is confident that if our code recycled memory more effectively on our Google Cloud instance, our neural network would be able train itself to be a good konane player. We unfortunately did not have the resources necessary to train the neural network, however given enough memory and time KonaneZero could produce a neural network which played konane with a high win rate on any given board size. The process of using AlphaZero code taught our group a lot about the process of training a neural network, optimizing gameplay, etc. While our group would have undoubtedly spent about one-tenth of the time on this project had we never touched the AlphaZero code, this experience of repurposing this code taught us a lot about troubleshooting with unfamiliar code, the system level operations of code execution, and neural network training.

***Implementing Min-Max***

With less that twenty-four hours before the tournament, our team repurposed our Artemis connector code to utilize a min-max search rather than the neural network. Luckily for the group, one of our team members had created both board and game objects before we had settled on initially repurposing AlphaZero. This meant we could repurpose the Artemis connector code to use these files rather than the KonaneZero objects.

**Code Documentation**

This section will document the two new files used for using a min-max search in the konane AI. These files use a similar board object to that which was described before but utilize a different search function and state evaluations for finding optimal moves.

Board.py

The Board object contains in Board.py maintains all functionality for modifying the state of the board. This includes initializing a full board, removing a single piece, and moving a move of any given jump length.

The Board object initializes itself using 0s, 1s, and a blank to represent the black, white, and empty spaces respectively. This is identical to how the Board object from the KonaneZero was initializes, just with different piece values. The Board object also initializes with a set dimensions for the board which are preset to 18 for the purposes of this project. This size is used only in the for loops that fill up the board initially and iterate over it to print the full board.

The removePiece function is extremely simple and merely and will take a tuple containing the coordinates of a piece and set the respective location to a blank spot.

The movePiece function will take two tuples, representing the start and end coordinates of a move, and clear all pieces between their linear positions. The function accomplishes this by checking to see which cardinal direction the move is made in and then iterating over the positions between them. The function initially will remove every piece between the start and end positions after which point it will set the location at the end position to the current player’s piece.

The Board object also re-defines the print statement for itself in a legible format. A Board object will print with vertical dividers between columns and horizontal dividers before and after printing the board itself. This makes reading an individual board easier as well as easing legibility when multiple boards are printed sequentially.

Konane.py

The Game object in Konane.py maintains the functionality to search for and choose moves for the AI to make during gameplay. The Game object maintains a Board object as well as which player it is. There are several other variables initialized in the Game object however these are not relevant for this discussion of the code.

To make a move, the Game object uses the computer\_move function. This function calls a min-max search which will return the optimally evaluated move. The minimax() function utilized follows a standard procedure for searching a game space. The minimax search uses two heursitics that can be manually changes: first, the ratio of number of pieces the current player has over the number of pieces the opponent has and; second, a heuristic found from a research paper by Darby Thompson (see reference 2) which evaluates the ratio of the current players number of available moves over the opponent’s number of available moves. The minimax search will evaluate every possible game state it finds using the generate\_successor\_states() function and return the one found to have the best evaluation.

The generate\_successor\_states() function will iterate over every possible board state it finds in the generate\_legal\_moves() function and generate a list containing Game objects for each of these board states. The function generate\_legal\_moves() will begin by calciulating the number of blank spaces on the current board. If there are no blank spaces it will return all possible first moves and if there is only one blank space it will return a list of all possible second moves. Beyond this the function looks for all normal jump moves that can be made. The function performs this search in an identical manner to what was utilized in KonaneLogic.py, so this will not be discussed in depth again.

**Repurposing Artemis Connector**

After switching from using a AlphaZero to a min-max search, the Artemis connector had to be modified to use the new objects for gameplay. During the initial development of the Artemis connector, our team was still working with the KonaneZero system. This meant that when we decided to make the transition from KonaneZero to a min-max search, the protocol for making moves and modifying the locally maintained board object had to be changed as well. As was previously mentioned, the code used during this transition was from the work one of the team members had done very early on in development. The Game object had been designed to play against itself which meant that it was assigned a player value on creation, and that after making a move it would change which player it was. For the purposes of our AI, we obviously only wanted to play one-side of the game. This meant that each time we called for the AI to generate a move, we have to reset the current player to what it was assigned by the server. The Game object also modifies its board state immediately following finding an optimal move. This was opposed to the server connector’s original protocol which followed waiting for the server to report a received move before modifying the board state. There are more minor changes that had to be made to the connector in order to refit our needs. but we will not discuss all of them in this paper. Overall the changes made were not numerous or complex but required a strong understanding of what had been written weeks prior for different purposes.

**Tournament Results and Best Evaluation Functions**

About a half hour before the start of the tournament, our team was comparing heuristic functions and found that the ratio of current player’s piece count over the opponent’s was outperforming the ratio of available legal moves. With such a short amount of time left until the tournament, our team decided to use the ratio of piece counts as our min-max heuristic. During our team’s first tournament game, we unfortunately lost by a multitude of moves. In our second game in the loser’s bracket we then lost again, however this time only by two pieces. At this point our team was confused; our AI had won almost 100% of all games during testing on this heuristic and it had now lost 100%. This prompted us to switch heuristics to the ratio of available moves for each player; our team figured that this heuristic could be better as it was only tested against other heuristics we had generated and not other teams AI.

After switching the heuristic function to the ratio of legal moves remaining, our AI won all but one game; the loss being against the AI which won the entire tournament. After changing the heuristic function, our team began asking other teams to run scrimmage matches to test out our theory that the ratio of legal moves remaining was a better heuristic. We began by playing the two teams we had originally lost to in the tournament whom we beat both of. At this point we began asking random teams in the room to play for a larger sample space of the AI’s performance. We found that of these games, we did not lose once. In some instances, the games were extremely close with only a single piece difference between players and in other cases our AI won by a larger margin. During our play with random teams, we encountered the team which had won first place in the winner’s bracket. To our surprise, our AI beat this team, showing our team that our AI was at a minimum just as good as the AI which won the winner’s tournament. The final game our AI played was against the team which won the entire tournament. While there seemed to be some sort of communication error between either or both of our server connectors, the winning team’s AI beat ours by an extremely large margin with only about a quarter of pieces being removed from the board before our AI had lost. Although our team was upset that this track record of the AI’s performance was not reflected in the tournament, our team is still extremely happy that we were able to self-validate that our AI as being about second best in the class.

**References**

1. AlphaZero Git Repository

<https://github.com/suragnair/alpha-zero-general>

1. Konane AI Neural Network Thesis Paper

<https://cs.brynmawr.edu/Theses/Thompson.pdf>